Dijkstra's in Disguise

Eric Jang

August 12, 2018

Keywords: *Graph Theory*, *Finance*, *Reinforcement Learning*, *Computer Graphics*

A weighted graph is a data structure consisting of some vertices and edges, and each edge has an associated cost of traversal. Let's suppose we want to compute the shortest distance from vertex u to every other vertex v in the graph, and we express this cost function as $\mathcal{L}_u(v)$.

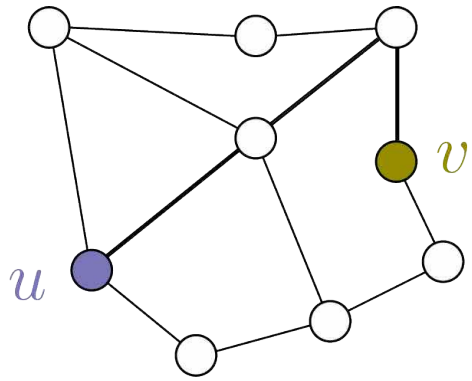


Figure 1: For example, if each edge in this graph has cost 1, $\mathcal{L}_u(v) = 3$.

Dijkstra's, Bellman-Ford, Johnson's, Floyd-Warshall are good algorithms for solving the shortest paths problem. They all share the principle of **relaxation**, whereby costs are initially *overestimated* for all vertices and gradually corrected for using a **consistent heuristic** on edges[1]. The heuristic can be expressed in plain language as follows:

[1] The term "relaxation" in the context of graph traversal is not be confused with "relaxation" as used in an optimization context, e.g. integer linear programs.

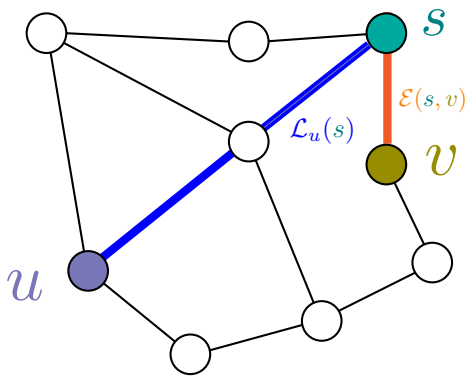


Figure 2: **Consistent heuristic**: The cost to reach v from u can be no greater than the cost to reach any of v's neighbors, plus the cost of traversing from that neighbor to the final destination v.

It turns out that many algorithms I've encountered in my **computer graphics**, **finance**, and **reinforcement learning** studies are all variations of this relaxation principle in disguise. It's quite remarkable (embarrassing?) that so much of my time has been spent on such a humble technique taught in introductory computer science courses!

This blog post is a gentle tutorial on how all these varied CS topics are connected. No prior knowledge of finance, reinforcement learning, or computer graphics is needed. The reader should be familiar with undergraduate probability theory and introductory calculus and willing to look at math equations. I've also sprinkled in some insights and questions that might be interesting to the AI researcher audience, so hopefully there's something for everybody here.

1 Bellman-Ford

Here's a quick introduction to Bellman-Ford, which is actually easier to understand than the famous Dijkstra's Algorithm.

Given a graph with N vertices and costs $\mathcal{E}(s,v)$ associated with each directed edge $s \to v$, we want to find the cost of the shortest path from a source vertex u to each other vertex v. The algorithm proceeds as follows: The cost to reach u from itself is initialized to 0, and all the other vertices have distances initialized to infinity.

The relaxation step (described in the previous section) is performed across all edges in any order for each iteration. The correct distances from u are guaranteed to have propagated completely to all vertices after $N-1$ iterations, since the longest of the shortest paths contain at most N unique vertices. If the relaxation condition indicates there are *still yet* shorter paths after N iterations,

it implies the presence of a cycle whose total cost is negative. You can find a nice animation of the Bellman-Ford algorithm here

Below is the pseudocode:

```
def bellman_ford(G, u):
  Lu = np.ones(G.num_vertices) * np.inf
  E = G.edge_weights
  Lu[u] = 0
  for i in range(G.num_vertices - 1):
    for (s, v) in G.edges:
      Lu[v] = min(Lu[v], Lu[s] + E[s, v]) # relax
  # Detect cycles
  for (s, v) in G.edges:
    if Lu[v] > Lu[s] + E[s, v]:
      print('Negative Cycle Found')
  print('No negative cycles')
  return Lu
```

2 Currency Arbitrage

Admittedly, all this graph theory seems sort of abstract and boring at first. But would it still be boring if I told you that *efficiently detecting negative cycles in graphs is a multi-billion dollar business*?

The foreign exchange (FX) market, where one currency is traded for another, is the largest market in the world, with about 5 trillion USD being traded every day. This market determines the exchange rate for local currencies when you travel abroad. Let's model a currency exchange's order book (the ledger of pending transactions) as a graph:

- Each vertex represents a currency (e.g. JPY, USD, BTC).
- Each directed edge represents the conversion of currency A to currency B.

An **arbitrage opportunity** exists if the product of exchange rates in a cycle exceeds 1, which means that you can start with 1 unit of currency A, trade your way around the graph back to currency A, and then end up with more than one unit of A!

To see how this is related to the Bellman-Ford algorithm, let each currency pair (A, B) with conversion rate $\frac{B}{A}$ be represented as a directed edge from A to B with edge weight $\mathcal{E}(A, B) = \log \frac{A}{B}$. Rearranging the terms,

$$\begin{aligned} \log \frac{A}{B} + \log \frac{B}{C} + ... \log \frac{Z}{A} &< 0 \\ \frac{A}{B} \cdot \frac{B}{C} ... \frac{Y}{Z} \cdot \frac{Z}{A} &< 1 \\ \frac{B}{A} \frac{C}{B} ... \frac{Z}{Y} \frac{A}{Z} &> 1 \end{aligned}$$

Figure 3: **Arbitrage**: Any cycle in an asset exchange graph having a product of conversion rates less than 1 is an arbitrage opportunity. We can re-write this as a negative cycle in a graph whose edge weights are in the form $\mathcal{E}(A, B) = \log \frac{A}{B}$.

The above algebra shows that if the sum of edge weights in a cycle is negative, it is equivalent to the product of exchange rates exceeding 1. The Bellman-Ford algorithm can be directly applied to detect currency arbitrage opportunities![2].

In my sophomore year of college, I caught the cryptocurrency bug and set out to build an automated arbitrage bot for scraping these opportunities in exchanges. Cryptocurrencies - being unregulated speculative digital assets - are ripe for cross-exchange arbitrage opportunities:

1. Inter-exchange transaction costs are low (assets are ironically centralized into hot and cold wallets).
2. Lots of speculative activity, whose bias generates lots of mispricing![3].
3. Exchange APIs expose much more order book depth and require no license to trade cryptos. With a spoonful of Python and a little bit of initial capital, you can trade nearly any crypto you want across dozens of exchanges..

Now we have a way to automatically detect mispricings in markets and end up with more money than we started with. Do we have a money printing machine yet?

[2]This also applies to all fungible assets in general, but currencies tend to be the most strongly-connected vertices in the graph.

[3]A simple form of bias being that retail traders like to buy things in integer dollar amounts. Another scenario: suppose an exchange rate of USD/BTC = 1/1. Let's also suppose ACME corporation has holdings in BTC and needs a lot of USD quickly to pay off some loans. Due to inadequate planning and human incompetence, ACME is so desperate to get this transaction filled quickly that rather than filling a bunch of small orders over the course of a few days, they are willing to pay a conversion ratio of 0.5 for a bulk order. As soon as ACME's order hits the order book (all the buy and sell orders), a trader could start with 1000 USD, fill ACME's order for 2000 BTC, and then trade it back to 2000 USD by 1) submitting a bunch of separate USD/BTC buy orders immediately or 2) slowly converting it back.

Not so fast! A lot of things can still go wrong. Exchange rates fluctuate over time and other people are competing for the same trade, so the chances of executing all legs of the arbitrage are by no means certain.

Execution of trading strategies is an entire research area on its own, and can be likened to crossing a frozen lake as quickly as possible. Each intermediate currency position, or "leg", in an arbitrage strategy is like taking a cautious step forward. One must be able to forecast the stability of each step and know what steps proceed after, or else one can get "stuck" holding a lot of a currency that gives out like thin ice and becomes worthless. Often the profit opportunity is not big enough to justify the risk of crossing that lake.

Simply taking the greedy minimum among all edge costs does not take into account the probability of various outcomes happening in the market. The *right* way to structure this problem is to think about edge weights being random variables that change over time. In order to compute the expected cost, we need to integrate over all possible path costs that can manifest. Hold this thought, as we will need to introduce some more terminology in the next few sections.

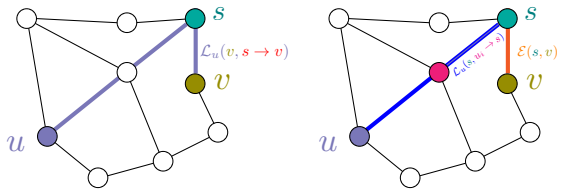
While the arbitrage system I implemented was capable of detecting arb opportunities, I never got around to fully automating the execution and order confirmation subsystems. Unfortunately, I got some coins stolen and lost interest in cryptos shortly after. To execute arb opportunities quickly and cheaply I had to keep small BTC/LTC/DOGE positions in each exchange, but sometimes exchanges would just vanish into thin air. Be careful of what you wish for, or you just might find your money "decentralized" from your wallet!

3 Directional Shortest-Path

Let's introduce another cost function, the **directional shortest path** $\mathcal{L}_u(v, s \to v)$, that computes the shortest path from u to v, where the last traversed edge is from $s \to v$. Just like making a final stop at the bathroom s before boarding an airplane v.

Note that the original *shortest path cost* $\mathcal{L}_u(v)$ is equivalent to the smallest *directional shortest path cost* among all of v's neighboring vertices, i.e. $\mathcal{L}_u(v) = \min_s \mathcal{L}_u(v, s \to v)$

Shortest-path algorithms typically associate edges with *costs*, and the objective is to *minimize* the total cost. This is also equivalent to trying to maximize *the negative cost* of the path, which we call $\mathcal{Q}_u = -\mathcal{L}_u(v)$. Additionally, we can re-write this max-reduction as a sum-reduction, where each $\mathcal{Q}_u$ term is multiplied by an indicator function that is 1 when its $\mathcal{Q}_u$ term is the largest and 0 otherwise.



$$
\begin{aligned}
\mathcal{L}_u(v, s \to v) &= \mathcal{E}(s,v) + \mathcal{L}_u(s) \\
&= \mathcal{E}(s,v) + \min_{u_i \in \mathcal{N}(s)} \mathcal{L}_u(s, u_i \to s) \\
Q_u(v, s \to v) &= \mathcal{E}(s,v) + \max_{u_i \in \mathcal{N}(s)} Q_u(s, u_i \to s) \\
Q_u(v, s \to v) &= \mathcal{E}(s,v) + \sum_{u_i \in \mathcal{N}(s)} \mathcal{L}_u(s, u_i \to s) p(u_i \text{maximizes } Q_u(s, u_i \to s))
\end{aligned}
$$

Figure 4: **Consistent heuristic (rephrased)**: The "directional shortest path cost" to the target node, through a neighbor of the target node, is the cost of the final step from the neighbor to the target, plus the cost of the shortest path to the neighbor node. Recursing into the neighbor node's own neighbors, note that the cost of the shortest path to the neighbor node is the min-reduction over all of the directional shortest paths for each of its neighbors. This is also equivalent to a maximization problem of negated costs.

Does this remind you of any well-known algorithm? If you guessed "Q-Learning", you are right!

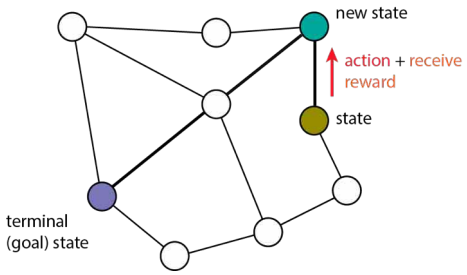
4 Q-Learning

Reinforcement learning (RL) problems entail an agent interacting with its environment such that the total expected reward R it receives is maximized over a multi-step (maybe infinite) decision process. In this setup, the agent will be unable to take further actions or receive additional rewards after transitioning to a terminal (absorbing) state.

There are many ways to go about solving RL problems[4], and we'll discuss just one kind today: **value-based** RL algorithms attempt to recover a value function $Q(s, a)$ that computes the maximum total reward an agent can possibly obtain if it takes an action a at state s.

[4] mathematically, all optimal control algorithms are just different faces of the same coin.

Wow, what a mouthful! Here's a diagram of what's going on along with an annotated mathematical expression.



$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a')$$

Figure 5: **Bellman Equality**: an agent takes action a at state s, receiving some reward $r(s,a)$ and arriving at a new state s'. The maximum obtainable total reward after taking a from s is the reward it just received, plus the maximum obtainable total reward starting from s', discounted by the probability that the agent randomly gets stuck and no longer can obtain rewards.

Re-writing the shortest path relaxation procedure in terms of a directional path cost recovers the Bellman Equality, which underpins the Q-Learning algorithm. It's no coincidence that Richard Bellman of Bellman-Ford is also the same Richard Bellman of the Bellman Equality! Q-learning is a classic example of dynamic programming.

For those new to Reinforcement Learning, it's easiest to understand Q-Learning in the context of an environment that yields a reward only at the terminal transition:

1. The value of state-action pairs (s_T, a_T) that transition to a terminal state are easy to learn - it is just the sparse reward received as the episode ends, since the agent can't do anything afterwards.

2. Once we have all those final values, the value for (s_{T-1}, a_{T-1}) leading to *those states* are "backed up" (backwards through time) to the states that transition to them.

3. This continues all the way to the state-action pairs (s_1, a_1) encountered at the beginning of episodes.

5 Handling Randomness in Shortest-Path Algorithms

Remember the "thin ice" analogy from currency arbitrage? Let's take a look at how modern RL algorithms are able to handle random path costs.

In RL, the agent's **policy distribution** $\pi(a|s)$ is a conditional probability distribution over actions, specifying how the agent behaves randomly in response to observing some state s. In practice, policies are made to be random in order to facilitate exploration of environments whose dynamics and set of states are unknown (e.g. imagine the RL agent opens its eyes for the first time and must learn about the world before it can solve a task). Since the agent's sampling of action $a \sim \pi(a|s)$ from the policy distribution are immediately followed by computation of environment dynamics $s' = f(s, a)$, it's equivalent to view randomness as coming from a stochastic policy distribution or stochastic transition dynamics. We redefine a notion of Bellman consistency for *expected* future returns:

$$Q(s,a) = \mathbb{E}_{a \sim \pi(a|s)} \left[r(s,a) + \gamma \max_{a'} Q(s', a') \right]$$

By propagating expected values, Q-learning allows for shortest-path algorithms to essentially be aware of the expected path length, and take transition probabilities of dynamics/policies into account.

6 Modern Q-Learning

This section discusses some recent breakthroughs in RL research, such as **Q-value overestimation**, **Softmax Temporal Consistency**, **Maximum Entropy Reinforcement Learning**, and **Distributional Reinforcement Learning**. These cutting-edge concepts are put into the context of shortest-path algorithms as discussed previously. If any of these sound interesting and you're willing to endure a bit more math jargon, read on – otherwise, feel free to skip to the next section on computer graphics.

Single-step Bellman backups during Q-learning turn out to be rather sensitive to random noise, which can make training unstable. Randomness can come from imperfect optimization over actions during the Bellman Update, poor function approximation in the model, random label noise (e.g. human error in assigning labels to a robotic dataset), stochastic dynamics, or uncertain observations (partial observability). All of these can violate the Bellman Equality, which may cause learning to diverge or get stuck in a poor local minima.

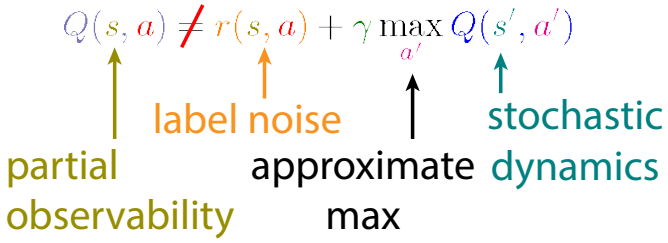


Figure 6: Sources of noise that arise in Q-learning which violate the hard Bellman Equality.

A well-known problem among RL practitioners is that Q-learning suffers from over-estimation; during off-policy training, predicted Q-values climb higher and higher but the agent doesn't get better at solving the task. Why does this happen?

Even if Q_θ is an unbiased estimator of the true value function, any variance in the estimate is converted into upward bias during the Bellman update. A sketch of the proof: assuming Q values are uniformly or normally distributed about the true value function, the Fisher–Tippett–Gnedenko theorem tells us that applying the max operator over multiple normally-distributed variables is mean-centered around a Gumbel distribution with a positive mean. Therefore the updated Q function, after the Bellman update is performed, will obtain some positively skewed bias! One way to deal with this is double Q-learning, which re-evaluates the optimal next-state action value using an i.i.d Q function. Assuming Q-value noise is independent of the max action, the use of a i.i.d Q function for scoring the best actions makes max-Q estimation unbiased again.

Dampening Q values can also be accomplished crudely by decreasing the discount factor (0.95 is common for environments like Atari), but γ is kind of a hack as it is not a physically meaningful quantity in most environments.

Yet another way to decrease overestimation of Q values is to "smooth" the greediness of the max-operator during the Bellman backup, by taking some kind of weighted average over Q values, rather than a hard max that only considers the best expected value. In discrete action spaces with K possible actions, the weighted average is also known as a "softmax" with a temperature parameter:

$$\texttt{softmax}(x, \tau) = \mathbf{w}^T \mathbf{x}$$

where

$$\mathbf{w}_i = \frac{e^{\mathbf{x}_i/\tau}}{\sum_{j=1}^{K} e^{\mathbf{x}_j/\tau}}$$

Intuitively, the "softmax" can be thought of as a confidence penalty on how likely we believe $\max Q(s', a')$ to be the actual expected return at the next time step. Larger temperatures in the softmax drag the mean away from the max value, resulting in more pessimistic (lower) Q values. Because of this temeprature-controlled softmax, our reward objective is no longer simply to "maximize expected total reward"; rather, it is more similar to "maximizing the top-k expected rewards". In the infinite-temperature limit, all Q-values are averaged equally and the softmax becomes a mean, corresponding to the return of a *completely random policy*. Hold that thought, as this detail will be important later when we discuss computer graphics!

This modification to the standard Hard-Max Bellman Equality is known as **Softmax Temporal Consistency**. In continuous action spaces, the backup

through an entire episode can be thought of as repeatedly backing up expectations over integrals.

$$Q(s,a) = r(s,a) + \gamma\tau \log \int_{a' \in A} da' \exp(Q(s',a')/\tau)$$

fubar

By introducing a confidence penalty as an implicit regularization term, our optimization objective is no longer optimizing for the cumulative expected reward from the environment. In fact, if the policy distribution has the form of a Boltzmann Distribution:

$$\pi(a|s) \sim \exp Q(s,a)$$

This softmax regularization has a very explicit, information-theoretic interpretation: it is the optimal solution for the **Maximum-Entropy RL objective**:

$$\pi^*_{\text{MaxEnt}} = \underset{\pi}{\text{argmax}}\, \mathbb{E}_\pi \left[\sum_{t=0}^{T} r_t + \mathcal{H}(\pi(\cdot|\mathbf{s}_t)) \right]$$

An excellent explanation for the maximum entropy principle is reproduced below from Brian Ziebart's PhD thesis:

> When given only partial information about a probability distribution, $\tilde{P}$, typically many different distributions, P, are capable of matching that information. For example, many distributions have the same mean value. The principle of maximum entropy resolves the ambiguity of an under-constrained distribution by selecting the single distribution that has the least *commitment* to any particular outcome while matching the observational constraints imposed on the distribution.

This is nothing more than"Occam's Razor" in the parlance of statistics. The Maximum Entropy Principle is a framework for limiting overfitting in RL models, as it limits the amount of information (in nats) contained by the policy. The more entropy a distribution has, the less information it contains, and therefore the less "assumptions" about the world it makes. The combination of Softmax Temporal Consistency with Boltzmann Policies is known as **Soft Q-Learning**.

To draw a connection back to currency arbitrage and the world of finance, limiting the number of assumptions in a model is of paramount importance to quantiatiative researchers at hedge funds, since hundreds of millions of USD could be at stake. Quants have developed a rather explicit form of Occam's Razor by tending to rely on models with as few statistical priors as possible, such as Linear models and Gaussian Process regression with simple kernels.

Although Soft Q-Learning can regularize against model complexity, updates are still backed up over single timesteps. It is often more effective to integrate rewards with respect to a "path" of samples actually sampled at data collection time, than backing up expected Q values one edge at a time and hoping that softmax temporal consistency remains consistent well when accumulating multiple backups.

Work from Nachum et al. 2017, O'Donoghue et al. 2016, Schulman et al. 2017 explore the theoretical connections between multi-step return optimization objectives (policy-based) and temporal consistency (value-based) objectives. The use of a multi-step return can be thought of as a path-integral solution to marginalizing out random variables occuring during a multi-step decision process (such as random non-Markovian dynamics). In fact, long before any of this Deep RL hype came around, control theorists have been using path integrals for optimal control to tackle the problem of integrating multi-step stochastic dynamics (`https://arxiv.org/pdf/physics/0505066.pdf`, `http://www.jmlr.org/papers/volume11/theodorou10a/theodorou10a.pdf`).

Once trained, the value function $Q(s, a)$ implies a sequence of actions an agent must do in order to maximize expected reward (this sequence does not have to be unique). In order for the Q function to be correct, it must also implicitly capture knowledge about the expected dynamics that occur along the sequence of actions. It's quite remarkable that all this "knowledge of the world and one's own behavior" can be captured into a single scalar.

However, this representaitonal compactness can also be a curse!

Soft Q-learning and PGQ/PCL successfully back up *expected* values over some return distribution, but it's still a lot to ask of a neural network to capture all the knowledge about expected future dynamics, marginalize all the randomness into a single statistic.

We may be interested in propagating other statistics like variance, skew, and kurtosis of the value distribution. What if we did Bellman backups over entire distributions, without having to throw away the higher-order moments?

This actually recovers the motivation of **Distributional Reinforcement Learning**, in which "edges" in the shortest path algorithm propagate distributions over values rather than collapsing everything into a scalar. The main contribution of the seminal Bellemare et al. 2017 paper is defining an algebra that generalizes the Bellman Equality to operate on distributions rather than scalar statistics of them. Unlike the path-integral approach to Q-value estimation, this framework avoids marginalization error by passing richer messages in the single-step Bellman backups.

Soft-Q learning, PGQ/PCL, and Distributional Reinforcement Learning are "probabilistically aware" reinforcement learning algorithms. They appear to be tremendously beneficial in practice, and I would not be surprised if by next year it becomes widely accepted that these techniques are the "physically correct" thing to do, and hard-max Q-learning (as done in standard RL evaluations) is discarded. Given that multi-step Soft-Q learning (PCL) and Distributional RL take complementary approaches to propagating value distributions, I'm also excited to see whether the approaches can be combined (e.g. policy gradients

over distributional messages).

7 Physically-Based Rendering

Ray tracing is not slow, computers are. – James Kajiya

A couple of the aforementioned RL works make heavy use of the terminology "path integrals". Do you know where else path integrals and the need for "physical correctness" arise? Computer graphics!

Whether it is done by an illustrator's hand or a computer, the problem of rendering asks "Given a scene and some light sources, what is the image that arrives at a camera lens?". Every rendering procedure – from the first abstract cave painting to Disney's modern Hyperion renderer, is a depiction of light transported from the world to the eye of the observer.

Production rendering technology has made tremendous strides in the last 20 years:

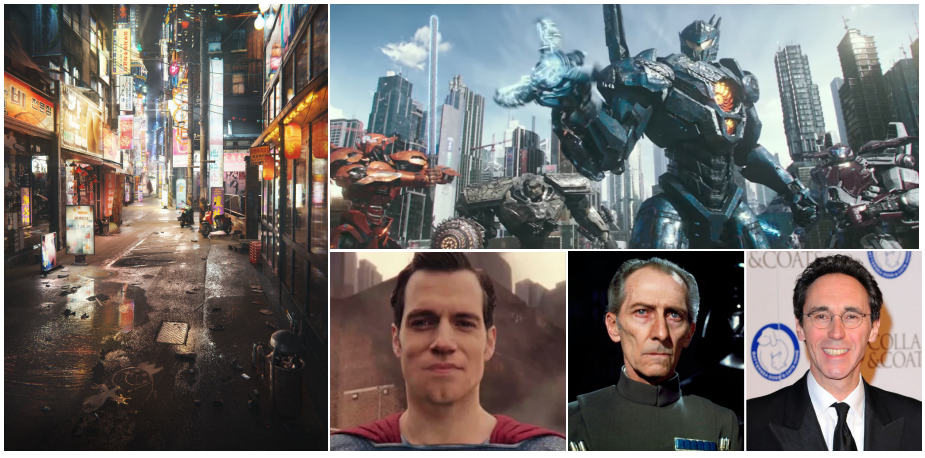


Figure 7: From top left, clockwise: Big City Overstimulation by Gleb Alexandrov. Pacific Rim, Uprising. The late Peter Cushing resurrected for a Star Wars movie. Remove Henry's Cavill's mustache to re-shoot some scenes because he needs the mustache for another movie.

Photorealistic rendering algorithms are made possible thanks to accurate physical models of how light behaves and interacts with the natural world, combined with the computational resources to actually represent the natural world in a computer. For instance, a seemingly simple object like a butterfly wing has an insane amount of geometric detail, and light interacts with this geometry to produce some macroscopic effect like iridescence.

Light transport involves far too many calculations for a human to do by hand, so the old master painters and illustrators came up with a lot of rules about how light behaves and interacts with everyday scenes and objects. Here are some examples of these rules:

- Cold light has a warm shadow, warm light has a cool shadow.
- Light travels through tree leaves, resulting in umbras that are less "hard" than a platonic sphere or a rock.
- Clear water and bright daylight result in caustics.
- Light bounces off flat water like a billiard ball - with a perfectly reflected incident angle, but choppy water turns white and no longer behaves like a mirror.

You can get quite far on a big bag of heuristics like these. Here are some majestic paintings from the Hudson River School (19th century).

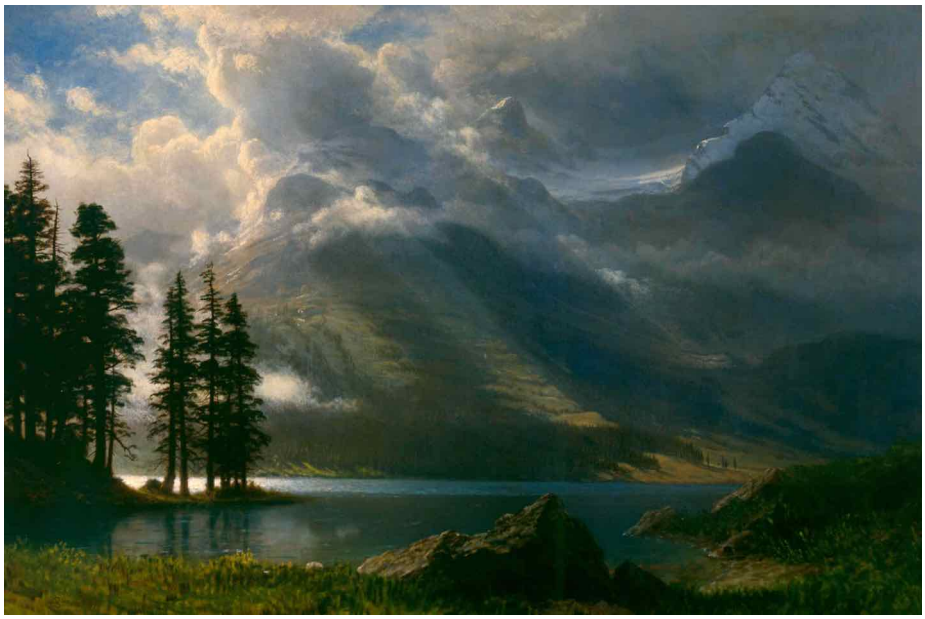


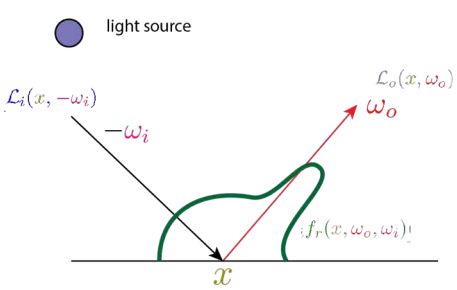
Figure 8: Albert Bierstadt, Scenery in the Grand Tetons, 1865-1870

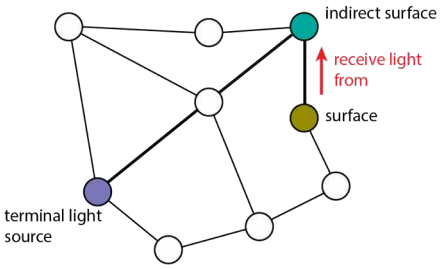
Figure 9: Albert Bierstadt, Among the Sierra Nevada Mountains, California, 1868

Figure 10: Mortimer Smith: Winter Landscape, 1878

However, a lot of this painterly understanding – though breathtaking – was non-rigorous and physically inaccurate. Scaling this up to animated sequences was also very laborious. It wasn't until 1986, with the independent discovery of the rendering equation by David Immel et al. and James Kajiya, that we obtained physically-based rendering algorithms.

Of course, the scene must obey the conservation of energy transport: the electromagnetic energy being fed into the scene (via radiating objects) must equal the total amount of electromagnetic energy being absorbed, reflected, or refracted in the scene. Here is the **rendering equation** explained in an annotated equation:





$$\mathcal{L}_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega d\omega_i f_r(x, \omega_o, \omega_i)(\omega_i \cdot n)\mathcal{L}_i(x, -\omega_i)$$

Figure 11: **Rendering Equation**: The outgoing radiance from a given point in a given direction is the emitted radiance in that direction, plus the total reflected radiance, which is made up of incoming radiance from every possible direction. Radiance from reflected, incoming light rays are attenuated by the probability of the light ray exiting in the direction of interest, given that it it entered the point from the incoming direction. It is also attenuated by the cosine of the facing angle of local surface geometry with respect to the incoming light ray. More generally, f_r may also be a function of wavelength λ and time t.

A Monte Carlo estimator is a method for estimating high-dimensional integrals, by simply taking the average over many independent samples of an unbiased estimator. Path-tracing is the simplest Monte-Carlo approximation possible to the rendering equation. I've borrowed some screenshots from Disney's very excellent tutorial on production path tracing to explain how "physically-based rendering" works.

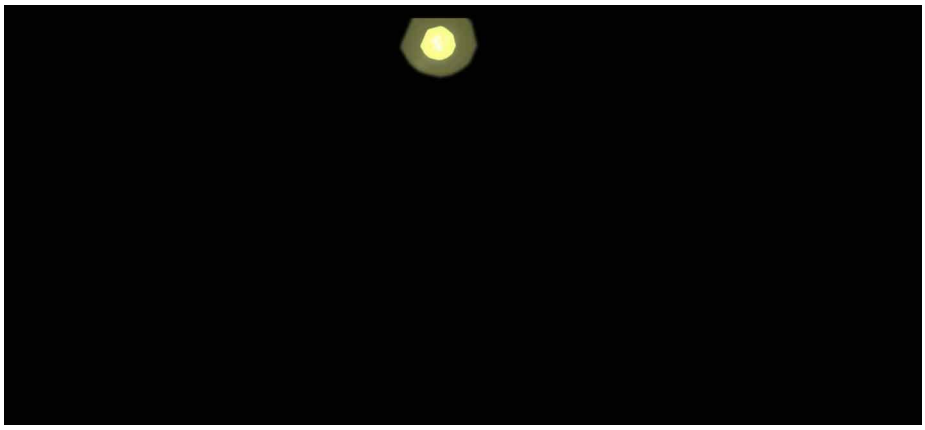


Figure 12: Initially, the only thing visible to the camera is the light source. Let there be light!

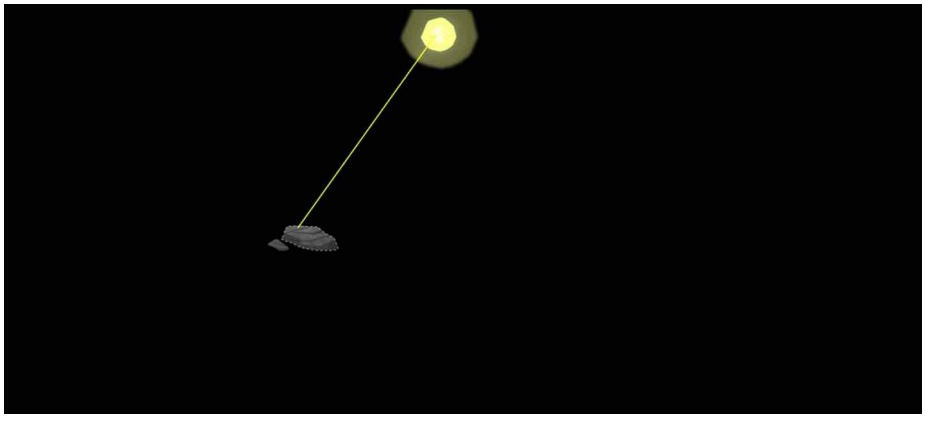


Figure 13: A stream of photons is emitted from the light and strikes a surface (in this case, a rock). It can be absorbed into non-visible energy, reflected off the object, or refracted into the object.

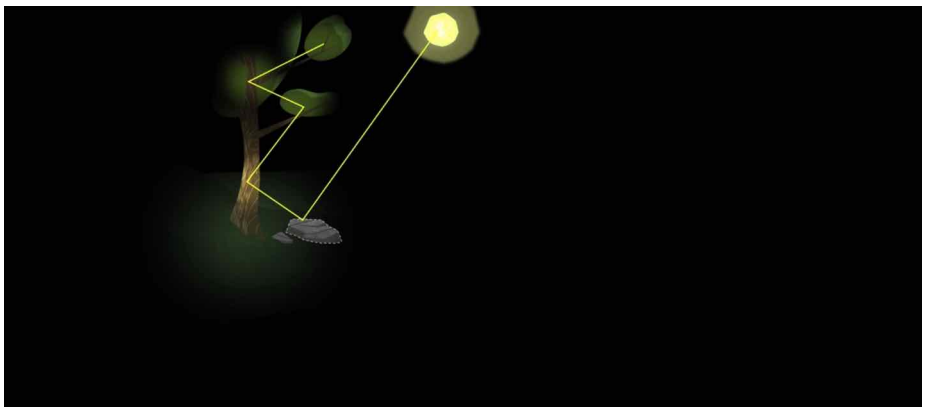


Figure 14: Any reflected or refracted light is emitted from the surface and continues in another random direction, and the process repeats until there are no photons left or it is absorbed by the camera lens.

Figure 15: This process is repeated ad infinum for many rays until the inflow vs. outflow of photons reaches equilibrium or the artist decides that the computer has been rendering for long enough.

This equation has applications beyond entertainment: the inverse problem is studied in astrophysics simulations (given observed radiance of a supernovae, what are the properties of its nuclear reactions?), and the neutron transport problem [5]. The rendering integral is also a **inhomogeneous Fredholm equations of the second kind**, which has the general form:

[5] In fact, Monte Carlo methods for solving integral equations were developed for studying fissile reactions for the Manhattan Project!

$$\varphi(t) = f(t) + \lambda \int_a^b K(t,s)\varphi(s)\,\mathrm{d}s.$$

Take another look at the rendering equation. Déjà vu, anyone?

Once again, path tracing is nothing more than the Bellman-Ford heuristic encountered in shortest-path algorithms! The rendering integral is taken over the 4π steradian's of surface area on a unit sphere, which cover all directions an incoming light ray can come from. If we interpret this area integration probabilistically, this is nothing more than the expectation (mean) over samples from a uniform sphere.

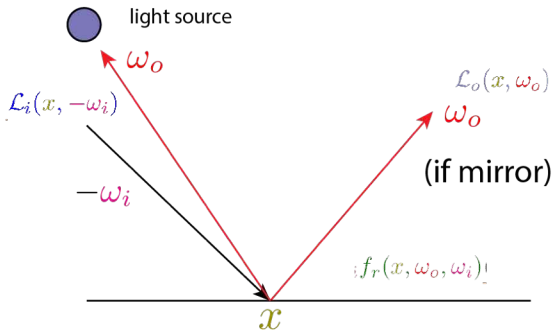
This equation takes the same form as the high-temperature softmax limit for Soft Q-learning! Recall that as $\tau \to \infty$, softmax converges to an expectation over a uniform distribution, i.e. a policy distribution with maximum entropy and no information. Light rays have no agency, they merely bounce the scene like RL agents taking random actions!

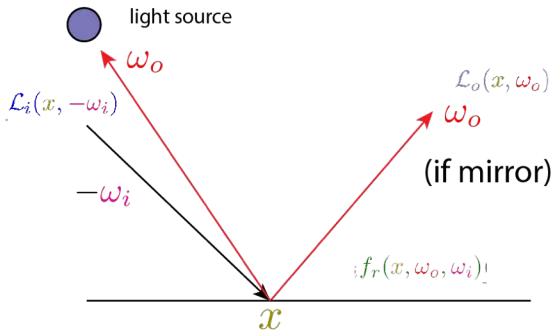
The astute reader may wonder whether there is also a corresponding "hard-max" version of rendering, just as hard-max Bellman Equality is to the Soft Bellman Equality in Q-learning.

The answer is yes! The **recursive ray-tracing** algorithm (invented before path-tracing, actually) was a non-physical approximation of light transport that assumes the largest of lighting contributions reflected off a surface comes from one of the following light sources:

1. Emitting material
2. Direct exposure to light sources
3. Strongly reflected light (i.e. surface is a mirror)
4. Strongly refracted light (i.e. surface is made of glass or water).

In the case of reflected and refracted light, recursive trace rays are branched out to perform further ray intersection, usually terminating at some fixed depth.

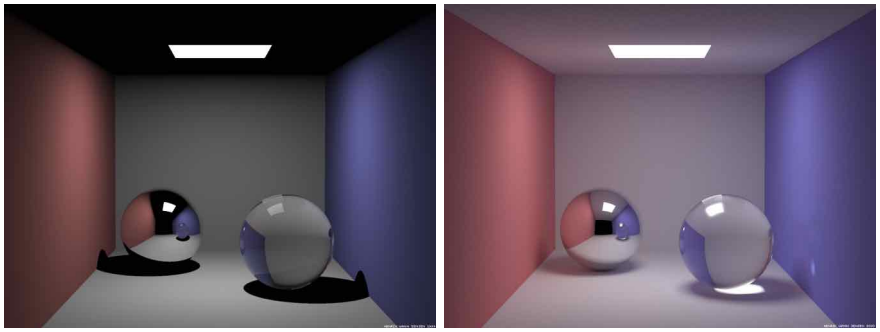




$$\mathcal{L}_o(x, \omega_o) = L_e(x, \omega_o) + \max_{\omega_i \in \Omega} f_r(x, \omega_o, \omega_i)(\omega_i \cdot n)\mathcal{L}_i(x, -\omega_i)$$

Because ray tracing only considers the maximum contribution directions, it is not able to model indirect light, such as light bouncing off a bright wall and bleeding into an adjacent wall. Although these contributions are minor in today setups like Cornell Boxes, they play a dominant role in rendering pictures of snow, flesh, and food.

Below is a comparison of a ray-traced image and a path-traced image. The difference is like night and day:



Prior work has drawn connections between light transport and value-based reinforcement learning, and in fact Dahm and Keller 2017 leverage Q-learning to learn optimal selection of "ray bounce actions" to accelerate importance sampling in path tracing. Much of the physically-based rendering literature considers the problem of optimal importance sampling to minimize variance of the path integral estimators, resulting in less "noisy" images.

For more information on physical rendering, I highly recommend Benedikt Bitterli's interactive tutorial on 2D light transport and Pat Hanrahan's book chapter on Monte Carlo Path Tracing.

8 Summary and Questions

We have 3 very well-known algorithms (currency arbitrage, Q-learning, path tracing) that independently discovered the principle of relaxation used in shortest-path algorithms such as Dijkstra's and Bellman-Ford. Remarkably, each of these disparate fields of study discovered notions of hard and soft optimality, which is relevant in the presence of noise or high-dimensional path integrals. Here is a table summarizing the connections:

	Hardmax	Softmax
Currency Arbitrage	$\mathcal{L}_v(v) = \min_{v \in \mathcal{V}(v)} [\mathcal{L}_u(s) + \mathcal{E}(s, v)]$	-
Reinforcement Learning	$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(s', a')$	$Q(s, a) = r(s, a) + \gamma \tau \log \int_{a' \in \mathcal{A}} da' \exp(Q(s', a')/\tau)$
Rendering	$\mathcal{L}_o(x, \omega_o) = L_e(x, \omega_o) + \max_{\omega_i \in \Omega} f_r(x, \omega_o, \omega_i)(\omega_i \cdot n)\mathcal{L}_i(x, -\omega_i)$	$\mathcal{L}_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega d\omega_i f_r(x, \omega_o, \omega_i)(\omega_i \cdot n)\mathcal{L}_i(x, -\omega_i)$

These different fields have quite a lot of ideas that could be cross-fertilized. Just to toss some ideas out there (a request for research, if you will):

1. There has been some preliminary work on using optimal control to reduce sample complexity of path tracing algorithms. Can sampling algorithms used in rendering be leveraged for reinforcement learning?

2. Path tracing integrals are fairly expensive because states and actions are continuous and each bounce requires ray-intersecting a geometric data structure. What if we just consider a lot of discrete points (e.g. scattered photons), pre-compute a visibility matrix, and use that to seed a final-gather render?

3. Path tracing is to Soft Q-Learning as Photon Mapping is to ...?

4. Has anyone ever tried using the Maximum Entropy principle as a regularization framework for financial trading strategies?

5. The selection of a proposal distribution for importance-sampled Monte Carlo rendering could utilize Boltzmann Distributions with soft Q-learning. This is nice because the proposal distribution over recursive ray directions has infinite support by construction, and Soft Q-learning can be used to tune random exploration of light rays.

6. Is there a distributional RL interpretation of path tracing, such as polarized path tracing?

7. Given the equivalence between Q Learning and shortest path algorithms, it's interesting to note that in Deep RL research, we carefully initialize weights but leave the Q-function values fairly arbitrary. However, all shortest-path algorithms rely on initializing costs to negative infinity, so that costs being propagated during relaxation correspond to actually realizable paths. Why aren't we initializing all function values to negative-valued numbers?

9 Acknowledgements

I'm very grateful to Austin Chen, Deniz Otkay, Ofir Nachum, and Vincent Vanhoucke for reading and providing feedback to this post. All typos/factual errors are my own; please write to me if you spot additional errors. And finally, thank you for reading!